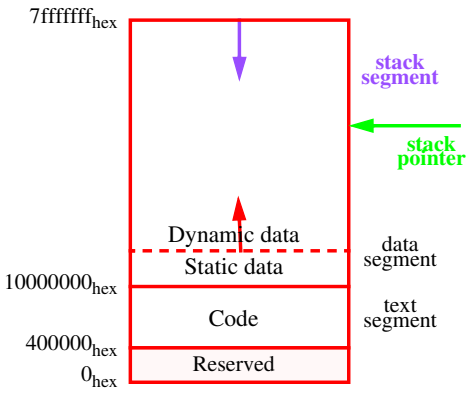
Subroutines

Why we use subroutines

- more modular program (small routines, outside data passed in)
  ⇒ more readable
  ⇒ easier to debug
- code reuse ⇒ smaller code space

Memory Usage

A software convention



stack segment:

- holds process-local storage managed in LIFO fashion
- automatically allocated by the operating system
- stack grows down (toward lower addresses) as data is put onto it
- **stack pointer $sp** (or $29) is automatically loaded to point to the last allocated location on the stack

Using the Stack

Stack operations:

- each requires 2 instructions
  - one to adjust the stack pointer
  - one to transfer the data onto/off the stack
- **push**:

```
subu $sp,$sp,4      # allocate a new word on the stack
sw  $t0,0($sp)      # push the contents of $t0
```

- **pop**:

```
lw  $t0,0($sp)      # pop the contents into $t0
addu $sp,$sp,4      # adjust the stack pointer
```

- the assembler recognizes a stack pointer operation & generates **`addui`**
- more efficient to allocate all the space you'll need at once
- often allocate & deallocate the same amount

Using the Stack

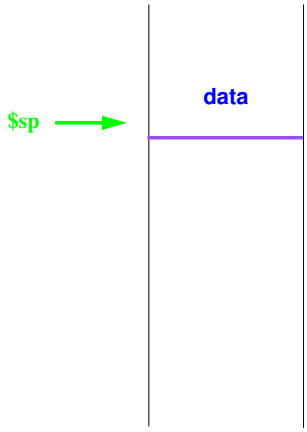
Example

saving values on the stack

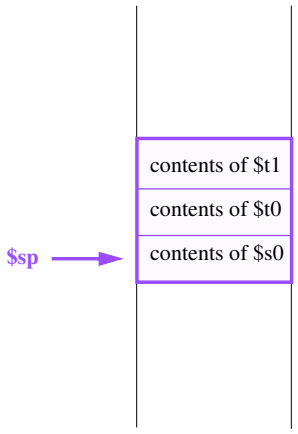
```
subu $sp,$sp,12
sw $t1,8($sp)
sw $t0,4($sp)
sw $s0,0($sp)
```

restoring values from the stack

```
lw $t1,8($sp)
lw $t0,4($sp)
lw $s0,0($sp)
addu $sp,$sp,12
```



- allocate all the stack space you will need for a group of data
- the stack locations are "above" where the stack pointer is pointing, so the displacements from $sp will be positive



Procedure Calling Convention

Protocol between the caller & the callee:

- who saves which registers
- how parameters are passed: in registers? on the stack? both?
- where the return address is located

Why have a calling convention?

- caller & callee can interact correctly & efficiently
- can have subroutines that are written in different languages & compiled with different compilers

Convention implemented by a division of labor between the hardware & software

- hardware:
  - performs simple instructions
  - has dedicated registers
- software:
  - controls the sequence of instructions for calling & returning
  - determines which register points to the stack
  - the rules of the protocol

Each architecture has its own protocol(s)

Why Have Two Classes of Registers?

Caller knows what registers it needs after the procedure call.

Callee knows what registers it is going to use.

Therefore divide the saving responsibilities to reduce **register spilling** (storing registers in memory because you need to use them for something else)

Caller-saved registers (**t**'s): not preserved across procedure calls

- caller saves them if it wants to use them after the procedure call
- if it doesn't need them later, it doesn't have to save them
  - so a caller uses them for short-lived values
- callee knows it can use these registers without saving them
- **savings:** caller-saved registers only spilled if the caller uses them after the procedure call

Callee-saved registers (**s**'s): preserved across procedure calls

- caller doesn't save them even if they are used after the return
  - a caller uses them for long-lived values
- if the callee needs to use them, it must save them first
- **savings**: callee-saved registers only spilled if the callee uses them

MIPS Procedure Calling Convention

Caller

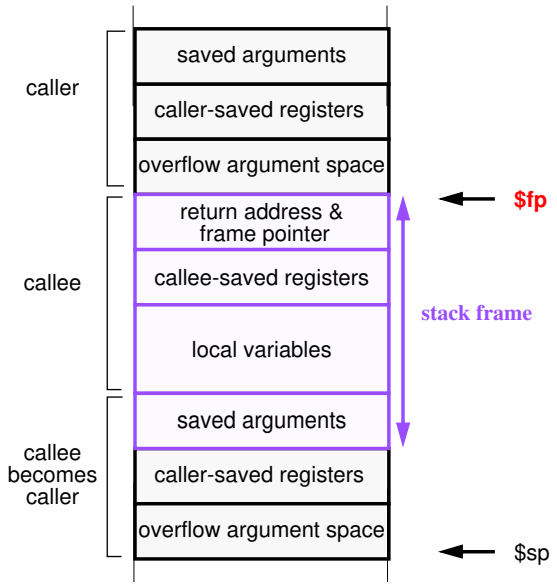
- saves the "a" registers on the stack
- saves the "t" registers on the stack: $t0 - $t9
- passes the arguments in $a0 - $a3
- saves additional passed values on the stack
- executes **`jal`** which puts the return PC in $ra

Callee

- determines the size of the stack frame & changes $sp to point to the end of the frame
  - **stack frame**: a region of the stack that holds all the data for a single procedure
    (also called procedure call frame, activation record)
  - `subu  $sp,$sp,frameSize`
- puts its return address & frame pointer ($fp) on the stack
- saves the "s" registers on the stack: $s0 - $s7
- sets $fp to $sp + stack frame size - 4
- puts its local variables on the stack
- if the callee calls a procedure, it becomes a caller

Neither modifies the other's portion of the stack

Stack Frame



frame pointer ($fp or $30):

- points to the first word of a stack frame
- value never changes: stable offset base for locations within the stack frame
- not always used

Example Call Sequence

Assume:

- arguments are in $t0 & $t1
- want to save caller-saved registers $t6 & $t7

```
move   $a0,$t0        # first argument
move   $a1,$t3        # second argument
subu   $sp,$sp,8      # adjust the stack pointer
sw     $t6,4($sp)     # save $t6 on the stack
sw     $t7,0($sp)     # save $t7 on the stack
jal    target
```

The first thing the callee will do
(assuming it does not have to save any $s registers)

```
target:   subu   $sp,$sp,4      # adjust the stack pointer
          sw     $ra,0($sp)     # save the return address
```

MIPS Procedure Return Convention

Return convention:

- put the return values in $v0, $v1
- restore the “s” (caller-saved) registers
- restores $fp & $ra
- pop the activation record by adding its size to $sp
- return to the caller by **`jr $ra`**

Example Return Sequence

Before returning, the callee will put results in $v0 & $v1 if needed & then:

```
lw      $ra,0($sp)      # load the return address into $ra
addu    $sp,$sp,4       # adjust the stack pointer
jr      $ra             # return
```

The caller will restore $t6 & $t7 and adjust the stack

```
lw      $t6,4($sp)      # restore $t6
lw      $t7,0($sp)      # restore $t7
addu    $sp,$sp,8       # adjust the stack pointer
```

Use of Registers & the Stack

Registers are used for:

- passing a small number of arguments (up to 4, $a0 - $a3)
- passing the return address to the callee ($ra)
- locating the beginning of the stack frame ($fp)
- returning values to a caller ($v0, $v1)
- keeping track of the top of the stack ($sp)
- returning function values ($v0, $v1)

Stack is used for:

- passing parameters if more than 4
- saving the caller's register parameters (a's)
  (might be used by the caller after the procedure returns)
- saving the caller's registers that are going to be used by the caller after the procedure returns ($t's)
- local data for the callee
- the return address of the caller
  (in case the callee becomes a caller & calls a procedure)
- the frame pointer (same reason)
- any callee-saved registers the callee is going to use ($s's)